Application Development Domain

Technical Architecture

Appendix B

Java Coding Standards and Conventions

February 6, 2003

Version 1.3

Revision History

Date	Version	Description
07/01/2002	1.0	Initial Draft
10/30/2002	1.1	Modifications
12/11/2002	1.2	Added references to Sun Coding Standards
02/06/2003	1.3	Added Sun copyright reference

Table of Contents

1. Introduction

1.1 Purpose

This document was created to provide guidelines for developing Java applications for the State of Connecticut. This document is written with generic terminology in order to allow continuity or similarity among all Java applications.

1.2 Scope and Application

Rules are those coding standards that are "necessary and required" coding practices that have been agreed upon by members of EWTA Application Domain Team. Everyone is expected to follow these "rules".

Guidelines are "suggested" coding practices that have been written to recognize the need for individuality AND for common coding practices. The purpose of guidelines is to provide a framework upon which we can all create better code. However, the guidelines are not meant to impede engineering efforts when these guidelines are found to be in direct conflict with an individual's preference, so long as that preference is implemented consistently and is well documented. Finally, because we recognize that this opens the code up to individual stylist coding habits, it is important that these habits are well documented and will then become the basis for all other updates within the affected files, i.e. when in someone else's code do as they do.

2. Rules

All Java Code developed for the State of Connecticut will follow the Java Coding Conventions[1] published by Sun which is published on the Sun Web Site at: http://java.sun.com/docs/codeconv/ (also published as Addendum A to the Application Domain Architecture) with the following addendums:

2.1 State of Connecticut addendums

2.1.1 File Names

Additional file suffixes and names *as an extension to Sun's conventions section 2.2*:

File Type	Suffix
Java source.	.java
Java bytecode	*.class*
Properties	.properties
HTML file	.html

[1] Copyright 1995-1999 Sun Microsystems, Inc. All rights reserved. Used by permission for non-commercial use.

XML file	.xml
XML Schema	.xsd
Document Type Definition	.dtd
Java Server Page	.jsp

2.1.2 Package Name

As an extension to Sun's conventions section 3.1.2, the domain extension 'us.state.ct.' shall be used before any package created for the State of Connecticut.

2.1.3 Method Names

As an extension to Sun's naming convention section 9, method names should begin with the first word of the name with a lowercase letter. Subsequent words in the name should have their first letter capitalized. The name of a method should use a verbNoun() or verbPrepositionNoun() form.

Example:

void **exampleMethodName()**

Accessor (getter) and Mutator (setter) methods should begin with "get" / "set".
Boolean getters should use "is", "has", or "can" as a prefix.

Example:

boolean **isEmpty()**
boolean **hasValues()**

2.1.4 Member Names

As an extension to Sun's naming convention section 9, member variable names should begin with a lowercase letter for the first word of the name. Subsequent words in the name should begin with an uppercase letter followed by lowercase letters. Member names should not start with underscore _ or dollar sign $ characters, even though both are allowed.

Example:

```
int  aClassMemberName;
```

2.1.5 Component Factory Names

A component factory is a public class that implements only static methods. These static methods are "Factory methods". Factory class names should include the word "Factory". Factory method names should start with the word "make."

Example:

```
public class WidgetFactory
{
   static Button makeButton(int aButtonType);
   static ListBox makeListBox();
   };
```

2.2 Syntactic Style Conventions

2.2.1 Import

As an extension to Sun's convention in section in 3.1.2, import statements shall include the fully qualified class name. The reason for including the class name is that the dependencies for the code are apparent. As a class is changed over time the decision of whether a package can be removed due to refactoring or bug fixes is easier to determine.

Example:

import **us.state.ct.test.base.Top**; ***//proper***
import **us.state.ct.test.base.*; *//bad practice***

2.2.2 Visibility

As an extension to Sun's programming practices section 10: In general, members of a class should be defined as private. This practice keeps encapsulation and allows implementation details to change without concern for coupling due to other class usage. However, there will be acceptable reasons to make a member, less restrictive. For example, a member that must be accessible to override behavior in a derived class where method access is not possible (due to the implementation) would be allowable. Additionally, access to constants for a class may be necessary and hence may not be private to the class. The key is to be able to defend the making of a member something other than private.

2.2.3 Variables

As an extension to Sun's Declaration section 6, variables should be defined just before their usage. This keeps blocking of code consistent and may reduce stack size for methods that have deep nesting. Only use a variable for one purpose. Reuse of variable names is a source for logic errors. Do not name a local variable the same as a member of the class. This hiding of a member is called shadowing and can lead to logic errors.

2.2.4 Methods

*As an extension to Sun's programming practices section 10, a*ccessor and mutator member functions should be provided for members of a class as per Java Beans coding practice. Exceptions to this rule may occur due to implementation needs (e.g., protection of an algorithm) or semantics (e.g., read-only class) of the class. Methods should avoid deep nesting structures though there may be exceptions when this is not possible. Methods should perform a single coherent function; avoid overloading a single method with parameters to perform different tasks based on that input. Consider the general functional areas of manager, implementor, helper, accessor, and mutator when trying to break a complex method into multiple methods.

2.2.5 Indentation

As an extension to Sun's indentation convention section 4, a single tab should be used instead of spaces when indenting. Each additional indention should correspond to a single tab. This tabbing convention is the preferred method, as opposed to C-style (K&R) tabbing convention, since it allows a file editor or IDE to specify the number of spaces to visualize for each tab present in the

file. In this way, each developer can view the file as they like while maintaining a consistent tabbing number.

2.3 Code Commenting

As an extension to Sun's comment section 5, all code must contain sufficient comments to generate usable JavaDoc. Comments must be included as follows:

2.3.1 Class Definition

The class definition is a brief description that documents the purpose and usage of the class. Items that should be covered in the documentation of the class include:

- Purpose statement
- Behavior statement
- Usage statement
- Author
- Version
- Caveats, if any. (Do not use unless there are some.)

Example:

```
/**
This class is for encapsulating connections to a database.
Other purpose statements…
This class maintains a pool of connections that are dispatched to the client.
Other behavior statements…
 This class should be access by using the static method getConnection().
Other usage statements…
        @author
@Version:
*/
```

2.3.2 Methods

Each method is preceded by a description in javaDoc format. Public, package and protected methods shall have a standard javadoc comment header. Method documentation should have the following parts:

- Purpose of method
- Implementation notes
- Parameter documentation
- Return documentation
- Exception documentation
- See tags if helpful

Example:

```
/**
   This method returns a connection from the pool of connections.
   @param inConnectString the database connect string returned on initialization.
```

```
    @return Connection The connection acquired.
    @exception SQLException If a database exception occurs.
    @see java.sql.Connection
  */
```

2.3.3 Members

A member of a class should be documented with the purpose of that member in the class. Optionally, constraints on the members use can be included.

3. Coding Guidelines

3.1 Recommendations

The following coding recommendations are presented to aid the developer in creating quality code. These recommendations may have exclusions and hence should be followed on a case-by-case basis. These items are presented in no particular order.

- In general, programs should have loose coupling of their component interfaces and high cohesion of elements within those components.
- Algorithms should be encapsulated within a component and called through a well-defined interface. This facilitates migration and evolution of the behavior represented by the algorithm.
- If a program is mainly composed of conditional statements such as If-then or switch, it should be reanalyzed for areas where modularization could be used to reduce the coupling exhibited by the conditional statements.
- Values within a loop should be calculated only once.
- If you can conceive of someone else implementing a class's functionality differently, define an interface, not an abstract class.
- Use static members only for the concept of class members and constants.
- Avoid returning the object of invocation unless method chaining is required.
- Other than for normal accessor/mutator methods, return types of methods should return derived results.
- Whenever reasonable, define a default (no-argument) constructor so objects can be created via Class.newInstance().
- Use method “.equals()” instead of operator == when comparing objects. In particular, do not use == to compare Strings.
- Use notifyAll instead of notify or resume.
- Assign null to any variable that is a reference to an object and is no longer being used. This allows the garbage collector to more efficiently cleanup objects.
- Avoid assignments (``=") inside if and while conditions.
- All float and double constants must have a decimal point with a digit on each side: 1.0 not 1., 0.3 not .3.
- Logical expressions that have more than one logical operator should have parentheses to indicate the operator priority.

- In the event that an exception must be handled and the program will take no adverse action, the catch block should not remain empty; instead, the block should contain a comment indicating the reasons for doing so.

3.2 Exceptions

3.2.1 General

Exceptions within JAVA have two forms: runtime and checked. Runtime exceptions are those, which are unanticipated and usually cause the system to need to recover to some steady state after occurring. Checked exceptions are exceptions which are recognized to be possible and are defined in the throws clause of a method. Checked exception shall be caught by programs and handled by using appropriate logging and recovery operations.

Exceptions should be mapped to domain specific exceptions when appropriate in the chain of exception handlers. For example, a database exception should be defined in terms of the operation that was requested by the user instead of returning the native SQL exception.

Here are some rules while making new exceptions:

- Create new exceptions on a broad scale. i.e. not to create an exception for each business rule, because generally a business rule violation would throw a "BusinessRuleException". But at the same time you can create new exceptions and put them in the proper hierarchy.
- All exceptions should at a minimum have a default constructor, a constructor which takes a String which contains the error message, and one that takes the string and a throwable for chaining of exceptions.

  Example:

  ****** Java Code for a custom exception *******

```
package us.state.ct.doit.exceptions;

/**
 * Custom Business rule exception to be thrown when a user attempts
 * to change a value that violates any business rule. This sample is
 * a "Broad" exception in that it can be used to report all types of
 * errors that occur when operating on business classes instead of
 * handling only one type of error such as defining a specific error
 * such as "LastNameException".
 *
 * @version 1.0 01/15/2003
 * @author Java Exception Class Author
 */

public class BusinessRuleException extends Exception
{
```

```
private Throwable cause = null;
/** the value the caller tried to set  */
private String newValue = "Not Reported";

/**
 * BusinessRuleException (Default constructor).
 *
 */
public BusinessRuleException()
{
   super();
}

/**
 * BusinessRuleException accepting a String indicating the error message.
 *
 * @param message String  Exception message
 */

public BusinessRuleException(String message)
{
   super(message);
}

/**
  * BusinessRuleException with a message and a throwable.
  * (The throwable allows for exception chaining).
  *
  * @param message  Exception message
  * @param cause    Cause of exception
  */

public BusinessRuleException(String message, Throwable cause)
{
   super(message);
   setCause(cause);
}

/**
  * BusinessRuleException with full details of the
  * value being set and the message indicating the reason for the exception.
  *
  * @param newValue String  The value that was trying to be set when the error
    occurred
  * @param message  String  Detailed message of the exception
  */
```

```
public BusinessRuleException(String newValue, String message)
{
   super(message);
   setNewValue(newValue);
}

/**
 * BusinessRuleException with full details of the
 * value being set and the message indicating the reason for
 * the exception and a throwable.
 * (The throwable allows for exception chaining)
 *
 * @param newValue String  The value that was trying to be set when the error
   occurred
 * @param message String   Detailed message of the exception
 * @param cause Throwable  Cause of exception
 */
public BusinessRuleException(
   String newValue,
   String message,
   Throwable cause)
{
   super(message);
   setNewValue(newValue);
   setCause(cause);
}

/**
 * Returns the cause.
 * @return Throwable
 */
public Throwable getCause()
{
   return cause;
}

/**
 * Returns the newValue.
 * @return String
 */
public String getNewValue()
{
   return newValue;
}

/**
```

```
    * Sets the cause.
    * @param cause The cause to set
    */
   private void setCause(Throwable cause)
   {
      this.cause = cause;
   }

   /**
    * Sets the newValue.
    * @param newValue The newValue to set
    */
   private void setNewValue(String newValue)
   {
      this.newValue = newValue;
   }

   /**
    * Override the toString method in order to provide
    * custom information about the entire exception.
    * @return String
    */

   public String toString()
   {
      return "Value in Error: "
         + getNewValue()
         + "\n"
         + "Message: "
         + getMessage();
   }

}
```

**** Sample class using the BusinessRuleException class defined above ****

```
package us.state.ct.doit.sample;

import us.state.ct.doit.exceptions.BusinessRuleException;

/**
 * Sample use of a custom exception in ordinary Java Code.
 * This code uses the various methods of the BusinessRuleException Class.
 *
 * @version 1.0 01/15/2003
 * @author Java Author
```

```
 */

public class EmployeeObject
{
   private String lName;
   private String fName;
   private int age;

   /**
    * Constructor for ExceptionHandleSample.
    */
   public EmployeeObject()
   {
      super();
   }

   /**
    * Returns the age.
    * @return String
    */
   public int getAge()
   {
      return age;
   }
   /**
    * Returns the fName.
    * @return String
    */
   public String getFName()
   {
      return fName;
   }

   /**
    * Returns the lName.
    * @return String
    */
   public String getLName()
   {
      return lName;
   }

   /**
    * Sets the age.
    * @param age The age to set
    * @throws BusinessRuleException if unable to set name
```

```
 */
public void setAge(int age) throws BusinessRuleException
{
   // This shows a sample of using the business rule exception to report
   // an error when setting an age value that is out of range.
   // When this exception is caught, the program will be able
   // to report the error message and the value sent to the method.

   if ((age < 1) || (age > 105))
   {
      throw new BusinessRuleException(
         String.valueOf(age),
         "Age must be in the range of 1 to 105");
   }
   else
   {
      this.age = age;
   }
}

/**
 * Sets the lName.
 * @param lName The lName to set
 * @throws BusinessRuleException if unable to set name
 */
public void setLName(String lName) throws BusinessRuleException
{
   // This shows a sample of using the business rule exception to report
   // an error when attempting to set the last name value that
   // cannot be empty. This new exception will only report an error.
   if ((lName.length() == 0) || lName.equals(null))
   {
      throw new
      BusinessRuleException("Length of Last Name must be greater than 0");
   }
   // This shows a sample of using the business rule exception to report
   // an error when attempting to set the last name value and the error is
   // something other than what we might expect.  In this case we create
   // a BusinessRuleException that contains the details of the Exception
   // that occurred.

   try
   {
      this.lName = lName;
   }
   catch (Exception e)
```

```
        {
            throw new BusinessRuleException("Unable to set Last Name", e);
        }
    }

    /**
     * Sets the fName.
     * @param fName The lName to set
     * @throws BusinessRuleException if unable to set name
     */
    public void setFName(String fName) throws BusinessRuleException
    {
        // This shows a sample of using the business rule exception to report
        // an error when attempting to set the last name value and the error is
        // something other than what we might expect.  In this case we create
        // a BusinessRuleException that contains the details of the Exception
        // that occurred.

        try
        {
            this.fName = fName;
        }
        catch (Exception e)
        {
            throw new BusinessRuleException("Unable to set First Name", e);
        }
    }

}
```